DESIGN of the MOSAIC ELEMENT

Chris Lutz
Steve Rabin
Chuck Seitz
Don Speck

5093:TR:83

Computer Science
California Institute of Technology
Pasadena CA 91125

The research described in this paper was sponsored by
The Defense Advanced Research Projects Agency
ARPA order number 3771
and monitored by the
Office of Naval Research
under contract number N00014-79-C-0597

DESIGN of the MOSAIC ELEMENT

Chris Lutz, Steve Rabin, Chuck Seitz, Don Speck

Department of Computer Science
California Institute of Technology
Pasadena, California 91125

ABSTRACT

The Mosaic element is a fast single chip computer designed to be used in groups for concurrent computation experiments. Each element contains a 16-bit processor, read-write storage, read-only store for a small initialization and bootstrap loading program, four input ports, and four output ports. The Mosaic processor, a highly structured design that achieves very good performance and density through innovations in its microcode, circuit techniques, and layout, is described in detail.

INTRODUCTION

Myriads of Mosaic elements can be connected together by their ports in a variety communication plans to form a family of specialized, high performance, concurrent, and programmable computing engines. In addition to its end use as a component for experiments with concurrent computing engines, the Mosaic element has been an interesting vehicle for numerous adventures in VLSI design, design tools, and testing. It includes experiments and innovations in its microcode, circuit techniques, and layout, with performance being a central objective throughout.

A Mosaic element with 4K bytes of read-write storage, approximately 140K transistors on a chip 4000 lambda square (6 mm square at 3 micron feature size), is sufficiently complex to have given our design tools a thorough workout, and have stretched our capabilities for laying out, verifying, and testing large structured designs.

The original models for this project were (1) Sally Browning's research on algorithms for a programmable tree machine[1,2,3] and (2) the "OM" described in Mead & Conway[4]. Mosaic started out as a tree machine element, but we have since come to see it as a building block for a variety of fine grain ensemble machines[5] with connection plans up to degree four, such as a tree, mesh, shuffle, chordal ring, or cube connected cycle. The influence of the OM2 on the processor datapath layout is apparent.

Several early attempts to lay out a much less ambitious processor with a 4-bit path to off-chip storage managed to break our design tools, and were thus indirectly the origin of the constraint solving composition and geometry tool Earl[6] used for the present design. A new processor with a 16-bit path to storage that could be placed on-chip was designed in 1982, sent to MOSIS in January 1983, and functioned essentially correctly, and at 7 MHz (4 micron feature size), on first silicon in February 1983. The processor design was subsequently augmented to include additional functions, to speed up the control PLA, and to incorporate the planned on-chip storage. It is this latest design that is described here.

TOP LEVEL VIEW

It appears that most of the silicon area in multiple instruction multiple data (MIMD) ensemble machines will be devoted to storage. In Cosmic Cube[7], a larger grain size machine at Caltech whose organization is otherwise similar to Mosaic, the fraction of the element complexity devoted to storage is about 75%. With the precondition that a complete Mosaic element fit on a single chip, and using today's MOSIS nMOS fabrication with 1.5 micron lambda (3 micron feature size) on chips 6 mm square, the complexity of today's Mosaic element is limited to 4000 by 4000 lambda, or 16 million square lambda (MSL). This area is apportioned with about 2.5 MSL for the processor and ports, 1 MSL for the pad frame, and 12 MSL (75%) for storage and its interconnect. The area allowed for the processor is quite small, less than 6 sq mm, or 9,000 sq mils.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

The storage is partitioned into several smaller arrays, as suggested by the analysis presented in section 8.5 of Mead & Conway[4]. Each array is 4096 bits, 64 by 64, organized to interface with the processor as 256 16-bit words. The densest read-write storage we understand how to make with MOSIS nMOS technology is based on a 3-transistor dynamic storage cell, which requires that this storage be refreshed periodically. This refresh function is accomplished in the microcode of the processor. The very small amount of read-only storage required for the initialization and bootstrap loader is implemented with a set of "maimed" RAM cells.

Thus the 16 MSL Mosaic has the floorplan shown in figure 1, but if more MSL were made available by a reduction in lambda, one could use this area to pack in more storage. While the processor is only 16% of the area of the chip, it represents about 90% of the design effort, so most of the following description concentrates on the processor.

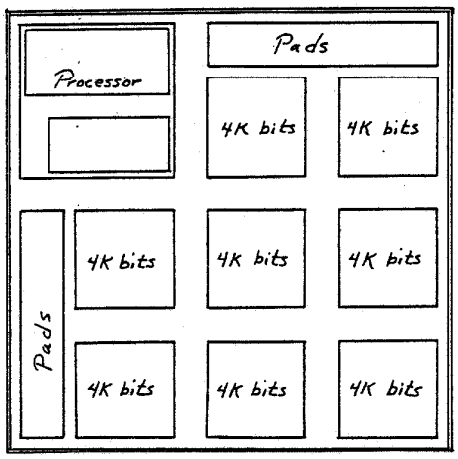


Figure 1: Mosaic Element Floorplan

The Mosaic element is synchronous, with 2-phase non-overlapping clocks supplied externally. The storage cycle, processor microcycle, and datapath operations occur in parallel in one clock cycle.

PROCESSOR ORGANIZATION

Figure 2 is a detailed block diagram of the processor, while figure 3 is the floorplan of the core of the processor, without the surrounding storage and pads. The processor has two principle components: a datapath/port block, and a controller. Each is a very dense, mostly metal-limited block of layout. The datapath/port block is functionally centered around the processor's single 16-bit internal data bus; it is controlled by signals issued by the PLA-based controller.

DATAPATH

The Mosaic datapath contains those parts of the processor that communicate over the internal data bus. These parts include sixteen general purpose registers, an ALU-shifter with associated flags, a memory address section, an interrupt counter, four input ports, four output ports, and an interface to the memory data bus and memory address bus. The ports are discussed in the following section. The functional blocks in the datapath are organized in a bit slice pattern, one bit of the bus running through each bit slice, with a bit slice pitch of 34 lambda. In the first clock phase of each cycle the bus is precharged and the ALU-shifter computes a new result. The second (last) clock phase is used for the bus transfer and the ALU carry chain precharge.

The ALU obtains operands from a pair of latches, called X and Y, that are loaded from the bus. The ALU result serves as input to the shifter, which uses pass gates to route correctly shifted data to the ALU-shifter output. The ALU is logically very similar to the ALU in the OM design[4], with a pair of function blocks and a precharged pass transistor carry chain. Although the ALU does not use carry lookahead, it is optimized to the extent that it is not in the critical timing path. An associated special purpose register, the Multiplier/Product, allows the processor to perform a multiply step in one microcycle. The multiply macroinstruction produces a 32-bit unsigned product in 20 microcycles.

The processor maintains four flags associated with the ALU/shifter. These are the familiar Z (zero result), N (negative result), V (two's complement overflow), and C (carry/not borrow). The controller cannot sense the values of the flags directly. Instead, a fixed 3-bit field in conditional branch macroinstructions specifies one of eight branch conditions. These three bits, as well as the values of the four flags, are inputs to a small PLA that produces one bit of output, the "flag condition". This bit is an input to the controller, which tests it when performing the conditional branch instructions. The branch condition codes were assigned carefully so the flag condition

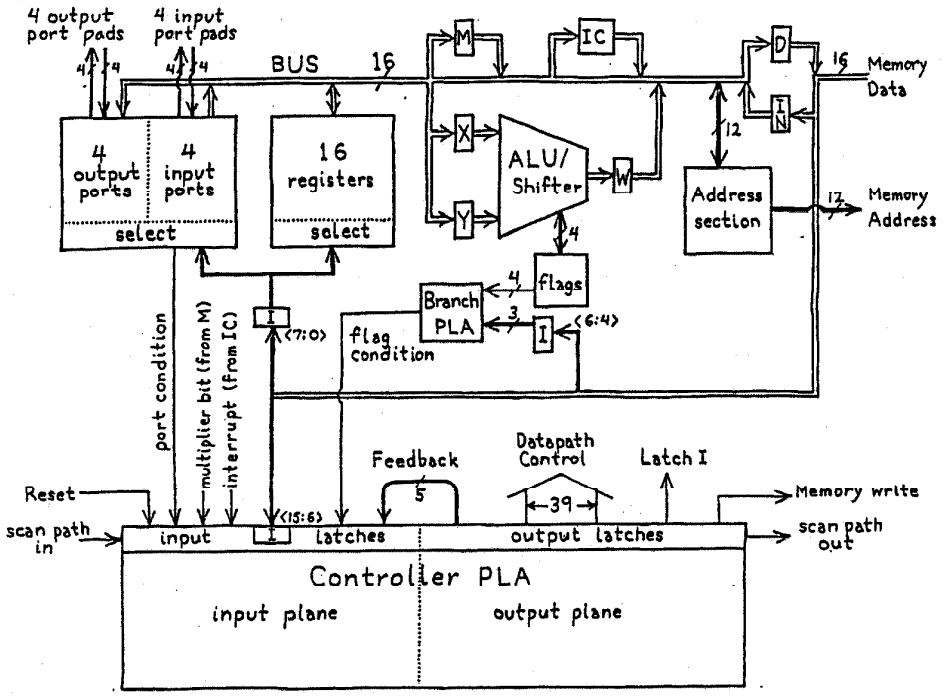


Figure 2: Processor Block Diagram

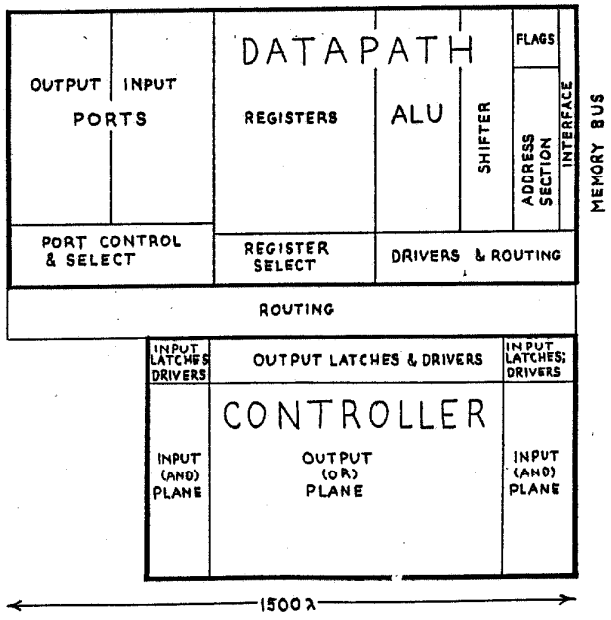


Figure 3: Processor Floorplan

PLA requires only seven implicants. Since the PLA is so small, it fits neatly next to the flags in the upper right corner of the processor, in a region formed by removing the top four bit slices of the address section.

On every microcycle the address section emits a new memory address onto the 12 memory address wires that come out of the right edge of the datapath. A 12-bit address is currently sufficient for the number of words of memory we can place on chip. The address generation section houses the program counter (PC) register, the current refresh address (RA) register, and an incrementer used with both registers. The controller guarantees that the RA is incremented and issued to the memory frequently enough, at least once every 8 microcycles, to keep the dynamic memory refreshed. Only memory cycles which would otherwise go to waste are used for refresh cycles.

The processor can generate timed interrupts using its interrupt counter (IC). The IC is a 10-bit register which counts down once per microcycle and causes an interrupt when it reaches zero. In order to guarantee interrupt service in bounded time, the port-wait states must be interruptible. Thus the side effects of any input or output instruction that cannot be completed immediately are reversed, and the instruction is refetched and restarted. Timed interrupts are useful for decoupling communications from processing, (*eg*, to implement automatic message routing, and to buffer large blocks of data) and to give the processor a sense of time (*eg*, for heuristic searches).

PORTS

Mosaic processors communicate with each other through their ports. Each processor has 4 input ports and 4 output ports. Connecting an output port of one processor to an input port of another processor forms a two-word fifo. Each output port is based on a parallel-in, serial-out shift register; each input port is based on a serial-in parallel-out shift register. The communication between input and output ports is bit serial.

Mosaic's implementation of the ports requires only a single wire, called the port link, to connect an input to an output port. When a port is not ready to perform a serial transfer, because it is an output port with no data or an input port with unremoved data, it clamps the port link to ground. On the microcycle when both ports are ready to perform a transfer, neither port grounds the port link and it is pulled up to VDD by an external pullup resistor. Both ports recognize this signal as the "start bit" of a transfer, much as in RS-232 data communications. The next 16 microcycles pass the data serially on the port link, and then the ports revert to the clamp-if-not-ready state.

This protocol allows multiple input ports to be connected to the same link: all input ports receive data from the output port beginning on the cycle when all the ports are ready. We didn't notice this feature until after we had running chips.

CONTROLLER

On every microcycle the Mosaic controller issues a new set of signals to control the datapath. The original plans for the controller assumed a rather conventional organization in which microcode words were fetched from a ROM, and a new ROM address was computed every microcycle by a conglomeration containing an incrementer, multiplexers, and other miscellaneous logic. This design was simplified when we realized that we could efficiently program a PLA to perform most of the original controller's function. An auxiliary PLA which controlled the ALU/shifter proved to be very troublesome because we could not find a placement for it that did not result in large wiring channels and expanses of white space. We finally eliminated the auxiliary PLA by learning to program the main controller PLA to perform its function. The controller became merely a PLA with latches.

In most microcircuit instruction processors the datapath is the more regular part, and the control the less regular part. In the Mosaic processor, the controller is even more regular than the datapath.

The controller has 20 inputs: 10 bits from the instruction register, the flag condition, the port condition, the multiplier bit from the multiplier/product register, the interrupt request from the interrupt counter, the processor reset, and 5 feedback bits (outputs from the controller clocked directly back to the controller input). The instruction register (I) holds the current macroinstruction and can be latched from the memory data bus on command from the controller. So little feedback state is needed because much of the state is held in the instruction register, and the sequences to implement macro instructions are short, typically 5 microcycles. (The shorter instructions are in practice executed more frequently, so

the average execution time is 4 microcycles.) Most of the 46 outputs from the controller go to clock-AND bootstrap drivers that drive control lines into the datapath. These outputs are effective during the microcycle following the microinstruction fetch.

INSTRUCTION SET

The tables in figure 4 summarize the macroinstruction set. All instructions are one word followed by zero, one, or two words of immediate data. In the first instruction word, the two 4-bit fields J and K can be used to specify one of the general registers. In some instructions, the K field may specify one of the ports or a branch condition instead.

There are two types of instructions: MOVEs and Arithmetics. MOVE instructions fetch an operand specified by a 3-bit MSOURCE field, and assign it as specified in the 4-bit MDEST field. The MOVEs incorporate subroutine linkage and branches: specifying an MDEST of PC performs a jump; an MDEST of PCF→@--R; X→PC performs a subroutine call by pushing the current PC on a stack, and then assigning it.

Arithmetic instructions fetch two operands, X and Y, as specified by the 3-bit MODE field. (The X and Y operands in fact correspond to the hardware registers X and Y at the input to the ALU.) Then they perform the operation specified by the 4-bit OP field, which requires computing some function of X and Y, and usually assigning a result as specified by the MODE.

Instructions that write to an output port wait until there is room in the fifo. Instructions that read from an input port wait until there is a word to read, and can optionally "advance" the port (remove the word from the fifo).

The richness of this instruction set is justified by the code compactness it offers in its environment of scarce on-chip memory.

MICROCODE

The speed, simplicity, and compactness of this design owe much to the realization that the controller need be nothing more than a PLA with latches. But a PLA is not merely sufficient; it is convenient and easy to program for an instruction set such as Mosaic's in which microinstruction sequences are short but heavily branched.

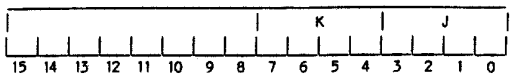
We chose to view each of the 120 implicants in the PLA as a word of microcode. More than one word of microcode can be active (that is, more than one implicant can be TRUE) in any given microcycle. Usually only one word is active at a time, but there are important exceptions. In these cases, the outputs are partitioned into disjoint sets, such that each active word has no TRUE outputs (transistors in the OR plane) outside its set. The effect of multiple active words used in this restricted manner is like that of multiple disjoint PLAs, but the physical layout retains the regularity of one PLA. In return for this self-imposed restriction, the absolute true/complemented sense of the individual outputs is irrelevant, the microcode assembler and assembly language is simpler, and the microcode is easier to understand.

A simple microcode assembler, written in SIMULA, reads the source microcode and assembles it into a runtime data structure. From here the assembler can output the code in any of several formats, including Earl source code. The assembler also contains an *ad hoc* register-transfer level simulator of the entire processor. This simulator served as an initial debugger for the processor design, and is still the initial testing ground for modifications in the processor and its microcode.

To illustrate some features of the microcode programming style and processor timing, the rest of this section is a blow-by-blow description of the execution of a sample macroinstruction. Figure 5 shows the assembly of the instruction ADD #7,R1,R2, the 4 microcode words required to execute it, and the behavior of various parts of the processor in the vicinity of these 3 cycles. This instruction adds immediate data 7 to the contents of register 1, and stores the result in register 2. The instruction executes in 3 microcycles, corresponding to the first, second, and last two microcode words (the last two are active simultaneously).

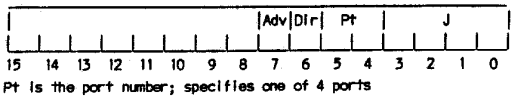
The tokens ".decode", ".get", and ".go" are mnemonics for feedback states; they appear both in the input conditions and in the next state outputs. The first microcode word, "DECODE:", is in fact the first word of every instruction. It becomes active any time the feedback state is ".decode", and no interrupt is pending ("Interrupt=0"). Previous microcode has ensured that a new macroinstruction was fetched on the previous cycle. Thus "DECODE:" latches it into

ALL INSTRUCTIONS:



followed by 0, 1, or 2 words of immediate value.

When K specifies a port:

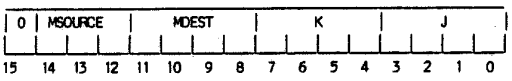


Pt is the port number; specifies one of 4 ports
Dir=0 for output port; Dir=1 for input port
Adv=1 to advance port if input port is read

KEY:
- Rn — is register number n.
- Rn++ — is register number n, incremented after reading.
- —Rn — is register number n, decremented before reading.
- val — is the immediate value.
- @z — is the memory word whose address is z.
- OutPort is output port number Pt.
- InPort is input port number Pt.
- A | B means concatenation of bit field A and bit field B.
- f<i> means the i-th bit of f.
- f<i:j> means i-th to j-th bits of f.
- FPC — is the flag/PC word: C | V | N | Z | PC

SPECIAL CASES: RESET: FPC -> @(-1); 0 -> PC
INTERRUPT: 0 -> IC; FPC -> @(-2); @(-3) -> PC

MOVE INSTRUCTIONS:



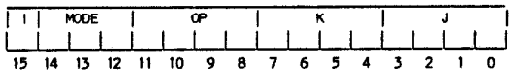
R means RK when MSOURCE is 0, 1, 2, or 3; R means RJ otherwise.

MSOURCE	X	MDEST	effect
0	RJ	0	X -> R
1	@RJ	1	X -> @R
2	@RJ++	2	X -> @R++
3	@(RJ+val)	3	X -> @(R+val)
4	val	4	X -> @—R
5	@val	5	X -> @val
6	InPort	6	X -> OutPort
7	IC	7	X -> IC
		8	IC -> R; X-> IC
		9	X -> PC
		A	X -> FPC
		B	X -> PC if flag condition K true
		C	X -> PC if flag condition K false
		D	X -> PC if port K is not ready
		E	X -> PC if port K is ready
		F	FPC -> @—R; X -> PC

FLAG CONDITIONS:

K	flag condition	K	flag condition
0	V [overflow]	4	Z [zero]
1	N [negative]	5	Z or N [<= zero]
2	¬C [Carry = 0]	6	Z or ¬C [unsigned <=]
3	N xor V [signed <]	7	Z or (N xor V) [signed <=]

ARITHMETIC INSTRUCTIONS:



MODE	Dir	X	Y	Dest	Assembly language example
0		RJ	RK	RK	ADD R1,R2
1		val	RJ	RK	ADD #val,R1,R2
2		@RJ	RK	RK	ADD @R1,R2
3		@val	RJ	RK	ADD @#val,R1,R2
4	0	RJ	0	OutPort	MOV R1,P1
5	0	val	RJ	OutPort	ADD #val,R1,P1
4	1	InPort	RJ	RJ	ADD P1,R1 [ADD P1+,R1 to advance]
5	1	val	InPort	RJ	ADD #val,P1,R1 [ADD #val,P1+,R1 to advance]
6		@RJ	RK	@RJ	ADD M @R1,R2
7		@val	RJ	@val	ADD M @#val,R1

All Arithmetic instructions modify the Z, N, and V flags.

OP	Instruction	Assembler Mnemonic	Effect		Carry flag modified?
0	INCrement	INC	X + 1	-> Dest	no
1	DECrement	DEC	X - 1	-> Dest	no
2	Arithmetic Shift Right	ASR	X<15>|X	-> Dest|C	yes
3	ROtate Right	ROR	C|X	-> Dest|C	yes
4	ROtate Left	ROL	X + X + C	-> Dest	yes
5	Logical Shift Right	LSR	0|X	-> Dest|C	yes
6	Rotate Nibble Right	RNR	X<3:0> | X<15:4>	-> Dest	no
7	ADD	ADD	X + Y	-> Dest	yes
8	ADD with Carry	ADDC	X + Y + C	-> Dest	yes
9	SUBtract	SUB	Y - X	-> Dest	yes
A	bitwise COMplement	COM	¬X	-> Dest	no
B	bitwise eXclusive OR	XOR	X exclusive or Y	-> Dest	no
C	bitwise AND	AND	X and Y	-> Dest	no
D	bitwise OR	OR	X or Y	-> Dest	no
E	COMpare	CMP	X - Y		yes
F	MULtiply	MUL	high word(X*Y)	-> RJ	no
			low word(X*Y)	-> RK	
			modify Z,N,V based on high word		

Figure 4: Diagram of the Complete Instruction Set

A macroinstruction, assembly language:

```
11: ADD #7,R1,R2
```

A macroinstruction, binary code:

```
10: ...                  [last word of previous instruction]
11: 1001 1001 0010 0001  [first word of instruction]
12: 0000 0000 0000 0111  [immediate value = 7]
13: ...                  [first word of next instruction]
14: ...                  [immediate value for next instruction, or
                             first word of instruction after next]
```

Source microcode for executing the macroinstruction:

The syntax for a source microcode word is:

```
word <mnemonic>: <inputs>                    :: <outputs>

word DECODE: .decode    Interrupt=0 ::  IN->I  RA++->A  RJ=> X Y M   .get

word #,J,K:  .get I= 1 0 0 1        ::  saveC  PC++->A  IN=> X       .go

word ADD:    .go I= 1 * * * 1 0 0 0 :: ALUONLY GP=86 Cin=0 NOshift setZNV setC
word ALU->K: .go I= 1 0 * *         :: NOALU   PC++->A    W=> RK     .decode
```

Processor timing in executing the macroinstruction:

micro-cycle number	microcode word(s) being fetched	microcode word(s) controlling processor	memory address being computed	memory address	memory data available	ALU function and bus transfer
	DECODE:	...	PC+1 = 12 (immediate value)	11 (ADD instr)	...	...
1	#,J,K:	DECODE:	RA+1 (new refresh address)	12 (immediate value)	ADD instr (latch into I register)	R1=> X,Y,M
2	ADD: and ALU->K:	#,J,K:	PC+1 = 13 (1st word of next instr)	(refresh address)	7 (immediate value)	7=>X
3	DECODE:	ADD: and ALU->K:	PC+1 = 14 (immed for next instr)	13 (1st word of next instr)	(refresh data)	X+Y->W W=>R2
	...	DECODE:	RA+1 (new refresh address)	14 (immed for next instr)	(1st word of next instr)	...

Figure 5: Example of a macroinstruction execution

the instruction register ("IN→I") at the start of the cycle. The controller has not had time to branch based on the new instruction, but the J and K fields will have arrived at the register decoder in time to select a register to drive the bus on that cycle; thus this microcode word fetches one of the registers to all of the destinations where it might be needed ("RJ⇒ X Y M"). This "register prefetch" saves a cycle in most instructions. It is too early to know what to do with the next memory cycle, so the microcode uses it as a refresh cycle ("RA++ →A", a macro for "RA→ INC Add1 INC→A A→RA").

The next microcode word "#,J,K:" is conditional on the first bit (1) and MODE bits (0 0 1) of the instruction register ("I= 1 0 0 1") and corresponds to an arithmetic instruction with an immediate value and a register as operands. In the complete microcode there is also a microcode sequence conditional upon each of the other possible MODE fields. The MODE in this example specifies operand X as an immediate value, which is obtained from the memory data bus via the memory data input buffer ("IN⇒ X"). The PC is then incremented past the immediate value ("PC++ →A") in order to begin fetching the next instruction. The next state ".go" indicates that all operands have been fetched and the code for the operative part of the instruction should take over.

The last two microcode words, "ADD:" and "ALU → K:" are active simultaneously and complete the macroinstruction. In the "ADD:" word the token "ALUONLY" indicates that this word specifies only ALU outputs (i.e. it has no transistors in the OR plane for other outputs) while "NOALU" in the "ALU→ K:" indicates that this word controls the rest of the outputs. The "ADD:" word instructs the ALU to add its inputs, X and Y, by specifying the appropriate Generate and Propagate codes ("GP= 86"), the carry-in ("Cin= 0"), and the type of shift ("NOshift"). The complete microcode contains similar words corresponding to the other arithmetic operations: subtract, increment, etc. They are independent of the MODE field of the instruction but dependent on the OP field ("I= 1 * * * 1 0 0 0", since OP code for ADD is 1000).

The "ALU→K:" word deposits the ALU output in register K ("W⇒ RK"). Other words, dependent on the MODE but independent of the OP code, handle the other possible destinations. Thus the orthogonality in the macroinstruction set, arithmetic OPs versus MODEs, is represented directly in the microcode. Note that only one microcode word, "ALU→K:", is needed to handle four MODE cases, since the MODEs have been carefully encoded so that one input condition ("I= 1 0 * *"), decodes all four cases. Careful encoding such as this throughout the instruction set has led to more compact microcode. In "ALU→K:" the PC is incremented and used as the memory address ("PC++ →A"), as it is in the last cycle of all instructions. This begins fetching the word after the next instruction, in case the next instruction takes an immediate value and needs to use it in its second cycle.

STORAGE

Although specialized semiconductor processes provide higher storage density than those suitable for the Mosaic processor, a processor with on-chip memory has many advantages over processor and memory in separate packages. These advantages include reduced volume, pin count, signal energy, and driver delay, resulting primarily from the integration of the memory bus into a single package.

For each storage bank, a two-bus three-transistor dynamic RAM memory cell is organized in 64x64 bits with a 16-bit word interface. All the banks operate in parallel to accomplish parallel refresh, and provide a read and pipelined write operation every processor microcycle (roughly 300 tau). Each memory access starts with a word-line access followed almost always by a refresh write to the same word-line on the next cycle. Because this write occurs in the same cycle as the next read, the storage control is somewhat subtle.

Mandating a write causes one of the 4 words read from the selected memory bank to be replaced with write data from the processor. This write data is written in the next cycle, in parallel with the next read. However, if the read is to the same word line as the pipelined write, stale data is accessed, and the subsequent write back to the word line would permanently store incorrect data. For this reason a write back is disabled on the second cycle after a write cycle. This form of pipelining imposes domain restrictions upon the microcode in that consecutive writes, refresh following write, and write followed by read to the same address will all fail. The first two conditions do not occur in the microcode, and the third condition occurs only by writing into the instruction stream.

CIRCUIT DESIGN

Some of the performance and layout simplicity of Mosaic is due to the simple clock-AND bootstrap driver shown in figure 6. It is used extensively and in several variations both in the processor and storage sections. In the processor, this clock-AND is used to produce control signals that are the logical-and of a PLA output and a clock. In the storage, the clock-AND is used so extensively in driving select and data lines that depletion transistors are completely absent.

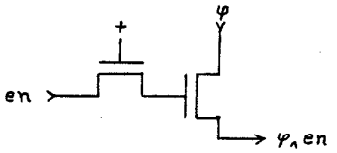


Figure 6: Clock-AND Circuit

Although referred to as a "driver," this clock-AND does not provide power amplification of the clock, but rather passes a replica of the "hot clock" input, whatever its HIGH voltage, to the output as gated by an enable signal of low energy. The clock signal typically switches between ground and 7 volts with VDD = 5 volts, but the chips also work correctly at reduced speed with 5 volt clocks. The delay and power dissipation of these clock-ANDs is almost negligible, and so the clock driving problem, together with the power dissipation usually required in control signal drivers, is exported to outside the chip where it can be dealt with using special driver circuits. This hot clock technique improves performance in pass structures, and also makes the performance much less sensitive to variations in the depletion threshold voltage than in conventional Mead-Conway designs. Precharge techniques are also used extensively in this chip, both to save power and for speed.

DESIGN TOOLS

The layout and verification was accomplished on a VAX-11/780 running Berkeley Unix with design tools written in MAINSAIL and C. Circuit design and optimization relied primarily on tau model calculations. SPICE was used to evaluate bootstrap effects, technology dependence, and critical timing paths.

The processor design is represented by 10,000 lines of code, interpreted by Earl[6], a constraint solving composition and geometry tool. Although the parts are composed in a rectangular bounding box discipline, the geometry internal to cells includes arbitrary angles and approximations of circular arcs, a form of "Boston geometry" that can be specified very easily in Earl. This unusual layout style is estimated to have reduced silicon area by 10% over 45-degree angle geometry, and by about 25% over Manhattan geometry.

For the design verification, the entire logic design was coded and simulated using the ternary switch level simulator MOSSIM[8] to verify logical correctness. After the layout was complete, raster extraction of layout using a Boston geometry circuit extractor produced a switch network that was used for MOSSIM II[9] simulations.

TESTING

First silicon for the Mosaic processor, received on 9 February 1983, 34 days after the CIF was submitted to MOSIS, was tested immediately and found to run code at a 7 MHz clock rate at room temperature. Subsequent processors fabricated using a faster process (still with a 4 micron feature size) ran at up to 11 MHz at room temperature.

Initial testing was accomplished by running the same code that had been used for switch level simulations. Subsequent testing using more extensive test programs discovered minor bugs that have been fixed in subsequent microcode. A scan path included in the original design between the datapath and controller was not used, although it might have been useful if anything had been seriously wrong.

Overall, our testing experiences have been quite similar to those reported by several other university groups, and point to two interesting development in testing for design verification. First, verification tools have advanced to the extent that nearly the entire design verification task is now accomplished before first silicon. Second, chips that are systems rather than components turn out to be simpler to test by placing them in their system environment than in a conventional tester, and the same tools that are used to program these systems serve to develop thorough tests of their function.

ACKNOWLEDGEMENTS

Chris Kingsley - Earl, Mike Schuster - Fsim, Howard Derby - early design, OM2 & GMP - ideas.

REFERENCES

[1] Sally A Browning, *Computations on a Tree of Processors*, Proceedings of the Caltech Conference on VLSI, January 1979, Computer Science, Caltech.

[2] Sally A Browning, *Hierarchically Organized Machines*, Section 8.4 in Mead & Conway[4].

[3] Sally A Browning and Charles L Seitz, *Communication in a Tree Machine*, Proceedings of the Second Caltech Conference on VLSI, January 1981, Computer Science, Caltech.

[4] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

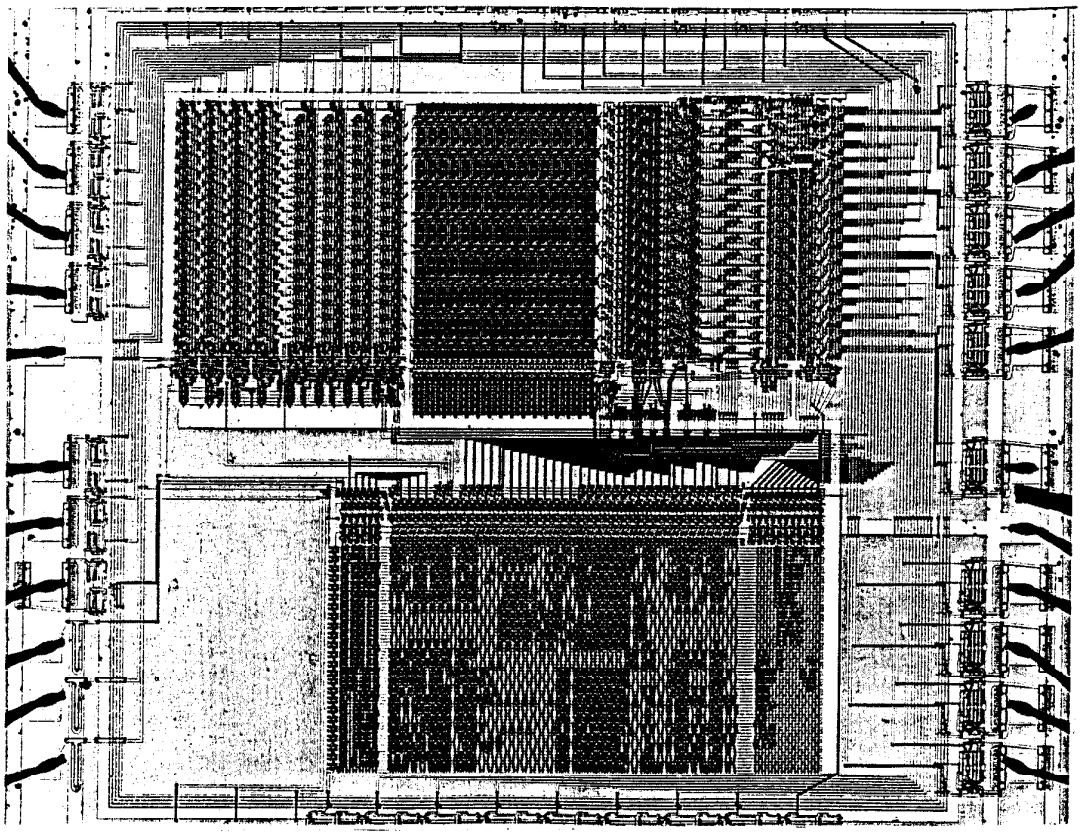
[5] Charles L Seitz, *Ensemble Architectures for VLSI*, Proceedings of the MIT Conference on Advanced Research in VLSI, January 1982, Artech Books, 1982.

[6] Chris Kingsley, *Earl: An Integrated Circuit Design Language*, Technical Report 5021, Computer Science, Caltech, June 1982.

[7] Charles L Seitz, *Experiments with VLSI Ensemble Machines*, Technical Report 5102, Computer Science, Caltech, October 1983.

[8] Randal E Bryant, *A Switch-Level Model and Simulator for MOS Digital Systems*, Technical Report 5065, Computer Science, Caltech, January 1983.

[9] R. Bryant, M. Schuster, D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Technical Report 5033, Computer Science, Caltech, March 1982.



Prototype Mosaic Processor